

(NASA-CR-180993) COMMENTS ON EVENT DRIVEN  
ANIMATION (NASA) 36 p Avail: NTIS HC  
A03/MF A01 CSCL 09B

N87-26566

Unclas  
G3/61 0085721

# RIACS

Research Institute for Advanced Computer Science

---

# **Comments on Event Driven Animation**

*Julian E Gomez*

**May 27, 1987**

**Research Institute for Advanced Computer Science  
NASA Ames Research Center**

**RIACS Technical Report 87.16**

## TABLE OF CONTENTS

|                                                 | Page |
|-------------------------------------------------|------|
| 1. Introduction .....                           | 2    |
| 2. Design points for an animation system .....  | 3    |
| 2.1. Interactive .....                          | 3    |
| 2.2. Speed .....                                | 4    |
| 2.3. Flexibility .....                          | 5    |
| 2.4. Extensibility .....                        | 6    |
| 2.5. Usability .....                            | 7    |
| 2.6. Habitability .....                         | 8    |
| 2.7. Overall .....                              | 9    |
| 3. Event Driven Animation .....                 | 9    |
| 3.1. The Display Function .....                 | 10   |
| 3.2. Definitions .....                          | 11   |
| 3.3. Interpolation .....                        | 12   |
| 3.4. Generality .....                           | 14   |
| 4. twixt .....                                  | 15   |
| 4.1. Input Methods .....                        | 15   |
| 4.2. Layering .....                             | 16   |
| 4.3. Objects .....                              | 17   |
| 4.4. Track Implementation .....                 | 19   |
| 4.4.1. Basic geometrical transform tracks ..... | 19   |
| 4.4.2. Hierarchy control tracks .....           | 20   |
| 4.4.3. Surface geometry track .....             | 23   |
| 4.4.4. Notes track .....                        | 24   |
| 4.5. Track Manipulation .....                   | 24   |
| 4.6. Record Structures .....                    | 26   |
| 4.6.1. Events .....                             | 26   |
| 4.6.2. Tracks .....                             | 28   |
| 4.6.3. Twerpers .....                           | 28   |
| 4.6.4. Comments .....                           | 29   |
| 5. Epilogue .....                               | 30   |
| 6. References .....                             | 32   |

## Comments on Event Driven Animation

*Julian E Gomez*

Research Institute for Advanced Computer Science  
NASA Ames Research Center

RIACS Technical Report 87.16  
May 27, 1987

**Event driven animation provides a general method of describing controlling values for various computer animation techniques. This report provides a definition and comments on generalizing motion descriptions with events. It also provides additional comments about the implementation of *twirt*.**

---

Work reported herein was supported in part by Cooperative Agreement NCC 2-387 between the National Aeronautics and Space Administration (NASA) and the Universities Space Research Association (USRA).

---

## Comments on Event Driven Animation

Julian E Gomez

Research Institute for Advanced Computer Science

May 27, 1987

### 1. Introduction

The most important design goal for an animation system is to not constrain the animator's imagination. The most serious problem with any animation system is the mass of detail required to produce animation.

We don't want a system to force a paradigm on the animator. In particular, it can't require physical laws, although it must be able to supply them when needed. A brief review of classical animation shows this point: although Wily Coyote falls in a fashion that may be related to  $d = 1/2at^2$ , it usually does not happen until he has been walking on air for a few seconds (the "Cartoon Laws of Motion").

No matter what method is used to describe motion, there is a large amount of data that needs to be specified. A system that provides only one type of movement will not provide the needed flexibility. As Wilhelms [23] points out, with a kinematics description the animator must experiment until the

motion looks right, and with a dynamics description the animator must experiment until the desired motion comes out.

The mathematics for computer animation and the techniques for building graphics software have been well explored. Higher level descriptions of animation will be the research area in the future. The last two decades have demonstrated that computer graphics can display animations with adequate form; now it's time to put some effort into constructing animations with content. Recent computer animations [2, 6, 11, 21, 22] show a definite move in this direction. This trend towards *character* animation will tax the capabilities of computer animation systems but produce more interesting animation.

## 2. Design points for an animation system

A topic related to the design parameters for an animation system is classification of animation systems, a subject treated by Zeltzer [25] and Gomez [9]. Both of these schemes rely on the motion specification mechanism for categorization. Qualifiers are also employed to describe practical aspects of the system, such as playback.

### 2.1. Interactive

An animation system should be interactive. It's hard to design pictures without looking at a picture of what's being designed and being able to change the picture and see directly what happens.

## 2.2. Speed

An ideal animation system would draw fancy color pictures in real time. Since this is impractical for the time being, the question becomes one of how much playback can be provided quickly. An acceptable answer is that as soon as the animator has finished adjusting something in the script, he can push a button and have the animation play back in real time. A few seconds delay for precalculation is acceptable; non real-time playback, however, is not. Whereas an animator can find something else to think about for the ten seconds or one minute of precalculation, it's difficult to appreciate motion when it is proceeding at the wrong rate. This is the way things used to be; cel animators wouldn't see any motion until perhaps the next day. In this day and age that's not a valid reason; a valid reason would be something like "this motion needs two minutes of Cray time to evaluate."

The problem is magnified on a multiprogramming system, where in addition to playing the animation at the wrong rate, the system will swap the animation system in and out of the execution queue, causing jerks in the animation.

We will call an animation system that provides acceptable playback an *online* system and designate it as being nice. With current technology, an online system will most likely provide a wireframe display.

### 2.3. Flexibility

The system shouldn't force the animator to use mechanisms she may not want to use. Sometimes an animator may want a linear spline, even with its attendant lack of continuity in the derivatives. As mentioned in the introduction, sometimes physics may be wanted and sometimes not. The point about flexibility is that the system should not force any motion mechanisms on the animator.

The subjects of splining and splines for computer animation have been discussed adequately in the literature, so these notes won't mention them other than in the description of the *twixt* implementation later.

A useful idea from the *scn\_asmblr* system [4] is the ability to substitute module names while interacting with the system. This *loose coupling* makes it easy for the animator to switch data resolution, change lighting algorithm, change anti-aliasing algorithm, change screen resolution, etc. Given this capability for changing parameters at whim and (relatively) immediately obtaining the new result, the animator is given extra ranges of expressive power. When it's trivial to change the way a picture is computed, the user will try those different ways, resulting in effects that may otherwise not have been attempted.



## 2.4. Extensibility

The basic reason for extensibility is that no matter what facilities the system provides, a need will arise for something else. This is especially true in a research and commercial production environments. Thus the system should include facilities for reconfiguring existing mechanisms or including new ones; it should be *extensible*.

An example of this can be found in the *emacs* editor. It provides a wealth of text operation functions, and the user can write subroutines using these operations to extend the power of the editor. A simple example would be a subroutine that transposes two lines; a more complex one would be an interactive e-mail handling repertoire. Once the user has written or borrowed such a routine, it is as easy to use as a built in *emacs* command, in addition to having the same interface. The point is that the user has the capability of modifying the system to his own desires without rewriting the program.

One way of using this extensibility is to have objects that carry their own behavior with them [13]. Humans, for example, can bend their elbows only so far. It would be nice to include this fact in the "human" abstraction. However, it would also be nice to be able to define a new type, say "human?," that has different or no restrictions on elbow movement.

This notion of dynamic use of the system means the animator can define his own movement criteria and use them in the animation system. This in

turn means the animator is effectively reconfiguring the system to his own needs for that animation.

In this context, object oriented programming is a generalization of extensibility. The advantages of object oriented programming extend themselves to any structured system, including one where the constituents are actors and motions rather than lines of source code.

Dynamic components require a rather sophisticated operating system. In particular, a program must be able to load code segments dynamically. Only LISP or Cedar [18] have this notion built into their design. Some efforts have been made towards bringing this attractive capability to UNIX, *e.g.* GEM [14].

## 2.5. Usability

The system should not be a keyframe system. Originally, a *keyframe* system was one which used *key frames* to control motion. It was designed to facilitate the way hand animation is built [20]. The key frames would be drawn by the animator and the system would interpolate between them. A number of such systems have been implemented [1, 3].

Recently, "keyframe" has been used in a more general sense to mean a system that interpolates between values, whether or not there actually are key frames. These are what Zeltzer calls *guiding* systems [25], indicating that the animator must explicitly describe the animation to be performed.

The system will provide splines to smooth out the animator's input.

The use of the terms *guiding* and *key parameter* is strongly preferred over *keyframe*, since the latter term implies there are key frames, in contrast to the first two, which do not. Since contemporary animation systems generally do not work with key frames, this accuracy is desirable.

Finally, the system shouldn't be an extension of a programming language. This forces the animator into a paradigm which has nothing to do with images. It's not necessary (neither is it prohibited) for an animator to know that a *for* loop is necessary to transform the vertices in a database, or that  $\cos^n$  is often used in illumination calculations. Furthermore, there is a strong possibility that the detail of dealing with a programming language will distract the animator from the animation.

## 2.6. Habitability

There are a number of other necessary features in an animation system contributing to its *habitability*, or how nice it is to work in the system. Examples are guarded exit (do not exit unless the script is saved or the user is sure); interactive exception handling (e.g. "File exists - do you want to overwrite it?"); help facilities. Defanti defined many habitability and extensibility requirements in GRASS [5].

## 2.7. Overall

A point not previously mentioned is that it may require more than one system to perform all these functions, with some sort of hierarchical arrangement between them [25]. This approach would provide different levels of complexity and the corresponding different levels of addressable detail.

## 3. Event Driven Animation

Event driven animation is an abstraction for describing animation. Rather than describing a specific animation technique, it describes a methodology for describing animation. It is not constrained to describing motion, but it is useful for constructing all aspects of an animation. It can be generalized to any level, thereby providing appropriate degrees of abstraction. A small scale event driven animation system can be implemented easily.

A fundamental concept when dealing with event driven animation is the idea that animation is not limited to moving things around; but also moving the color or the shape or the rate of change of the animation variables. The concept of event driven animation unifies all the different aspects of making an animation. The idea is that all display functions can be treated the same so that operations can be performed on any display function as easily as on any other, freeing the animator from having to use method  $m_1$  to deal with display function  $F_1$  and method  $m_2$  to deal with display function  $F_2$ . For

example, it's not acceptable for the animator to have to use key joint angles for arm motion and have to use inverse kinematics for leg motion.

Another way of putting this is that animation is not just getting from point A to point B using points C and D to help control a cubic spline; it's dealing with every aspect of making a picture and making the picture move. Thus the mechanisms for performing operations on anything should be similar.

This point can be qualified to a degree. It doesn't make sense to apply vector operations to a scalar value. However, the system should recognize the problem and deal with it, perhaps translating the request to something reasonable. Or the animator could have the options of configuring the system to attempt a translation, ignore the problem, or complain and ask for instructions.

### **3.1. The Display Function**

Consider some arbitrary display function: given some input parameters telling it how to operate, it will take data, process it, and output new values contributing to the picture. Common display functions include the basic geometric transforms such as translation, orientation, and scaling. Other display functions include color, transparency, surface geometry, whether or not to display, joint angles, etc.

It's readily apparent that the datatype required depends on the display function: color is a 3-vector, transparency is usually a scalar, orientation

is a 3x3 matrix, surface geometry is a dataset, whether or not to be displayed is a Boolean value, and a methodology for calculating a value is a procedure pointer. When one of these is used to control a display function, we will call it a *control value*.

Mentally we can translate any datatype into a vector of appropriate scalar values. Thus a matrix becomes a 9-vector of real numbers, a dataset becomes a matrix of 3-vectors which is in turn a  $3 \times n$  vector, a display flag becomes a 1-vector of Boolean values, and a procedure pointer is a pointer valued 1-vector. This point is academic, however, and is mentioned only for formality.

### 3.2. Definitions

The animation process requires specification of values for every frame of time for every display function implemented in the graphics system. For an arbitrary display function  $F$  we have a set of control values for it,  $\vec{v}_i$ . To each of these control value vectors we attach the time at which it is to be used; this construction of the control value and the time we call an *event*. The list of events describing the activity of  $F$  over the animation we call a *track*. The track is implicitly sorted in ascending order by event time; sorting should be implemented by the underlying software so the user doesn't have to do it.

Practically, it makes more sense to store events only when an input value for  $F$  changes, and use a splining technique to generate the inbetween values. Thus interpolation information must also be stored in the event: acceleration/deceleration information, splining method, *etc.* This will be discussed momentarily.

To access values within tracks, we can define an abstract function

$$E(\text{objects}, f, t)$$

where *objects* indicates a class of objects,  $f$  is the display function, and  $t$  is the time.  $E$  will return a value appropriate for that display function. The animation controller will have to evaluate the appropriate tracks to calculate that return value vector. The number of events necessary to do this will depend on the display function and the complexity of the splining method, *e.g.* a cubic spline requires four events to work with; a Boolean function requires only the closest preceding event.

Timing in an animation can be changed by changing the frame numbers in events. Track segments can be moved to change the time at which their animation occurs. Track segments can also be multiplied by a factor to expand or compress their length.

### 3.3. Interpolation

We begin to see a relation between events and curve generation. In fact, the values contained in the events are control points, the frame number is

the parameter of interpolation, and the animation for that display function is the result of the generated spline. Here the term *control values* is better than *control points*, to emphasize the fact that event values have arbitrary types, including some which cannot be splined.

There are a plethora of techniques for interpolating or approximating curves. Track animation relies on *patched* curves. Briefly, *patching* refers to the process of "gluing" together splined curves end to end, or surface elements side to side. Continuity in the derivatives across the boundaries, although desirable, is not required. Different interpolation functions can be used to achieve different patches, although the animator can certainly specify a single splining function for the entire duration of the track.

We must assume a track is at least piecewise continuous; otherwise  $F$  will be undefined at certain frames, a potential source of serious problems. We would also like the first derivative with respect to time,  $dF/dt$ , or  $\dot{F}$  (velocity) to be continuous; this will prevent sudden jumps in the output values from  $F$ . If the second derivative  $d^2F/dt^2$ , or  $\ddot{F}$  (acceleration) is also continuous, this will prevent sudden jumps in the rate of change of the input values to  $F$ . If both constraints are met then the output of  $F$  will be smooth and will change smoothly. See also Smith [17].

In order to calculate a frame in an animation, all tracks are evaluated for the given time. The collection of activity on all tracks inherently generates the animation.



Note that a change in interpolation functions is a change in value, and events can exist just for this purpose. Changes in velocity are also events, i.e. specifying values for  $d\mathbf{F}/dt$  instead of  $\mathbf{F}$  itself.

Keep in mind that interpolation schemes apply to any track, not just position. There is no reason why B-splines can't be used on color or transparency information; for continuity purposes, it's better if they are.

### 3.4. Generality

The level of abstraction for any track is tied to the intelligence of its *twerper* (interpolator). Position is trivial, rotation matrices are harder, surface geometries are even harder, object collision detection is yet harder, *etc.* The sophistication of the *twerper* is generally based on the amount of support code available and how much dynamism the operating system can provide.

One way of viewing different levels of tracks is to think of the higher levels compiling down to the lower levels. Just as a high level language is compiled down to a low level language, an abstract track can be compiled down to simpler ones. This allows the animator to deal with varying levels of abstraction, or with animation systems at different levels in Zeltzer's hierarchy. The unifying element among different tracks with different complexities is that the animator has provided certain values at certain points in time to control their behaviors.

Earlier it was mentioned that a control value could be a pointer to a procedure that controls the display function. This procedure would be invoked whenever the animation controller determines that it should be contributing to the calculation of the animation. This is somewhat analogous to "buttons" in Cedar [18, 19], which are modules invoked when a user clicks a button on the screen. In Cedar, part of the process of installing a new button on screen is to tell the window manager what procedure to invoke when the user clicks that button. In the same way, construction of these *inquisitive* events lets the track manager know what procedure to use when evaluation of a track is necessary. This facility is the most powerful aspect of event driven animation, as it allows dynamic control of the animation, where display function controllers can use the current values of other tracks in determining their own values, and thus respond to environmental parameters.

#### 4. *twixt*

*twixt* is integrated into the OSU image generation pipeline [24]. This gives the animator a unified environment for dealing with animation and image production.

##### 4.1. Input Methods

As described previously [7], there are a number of ways to describe values to *twixt*. The fancier the display device the user is working, the better

these input methods are. Where the input device provides only a limited number of inputs (*i.e.* a bank of control dials), *twixt* provides ways of dynamically changing the assignment of each input device to a control mechanism.

#### 4.2. Layering

The approach to designing animation in *twixt* is *layering*, where the animation is built up in layers of motion. Analogies can be drawn to cel animation, where a frame is built up of a number of cels lying on top of each other. In *twixt*, however, the layers are not pieces of picture, but pieces of motion.

An animator may labor for some time on one particular part of the animation, say the arm of a baseball pitcher throwing a ball. Then he may switch to the ball and work on that. This might be interspersed with quick returns to the arm to perfect some aspect of its motion. It might also be interspersed with work on the snap of the pitcher's head. No commands are required to switch context; the animator is carrying the context in his mind, and the naming scheme in *twixt* allows different ways of specifying the context of an action.

The intent of this approach is that it allows the animator to concentrate on one theme for some time, until he is ready to concentrate on another. It also allows the animator to instantly return to any previous activity in

order to modify it. This allows quick implementation of flashes, where the animator remembers or thinks of something that should be done to a sequence already worked on.

#### 4.3. Objects

*twixt* supports the common practice of constructing object hierarchies, *i.e.* of inserting subtrees into trees to express hierarchical relationships. Thus a scene is actually made of a *forest*[10] of trees. However, the relationships that can be expressed between nodes cover a broader range than that usually available, including operations that cannot be expressed as matrix products. A later section will elaborate.

One nice feature in *twixt* is the way the animator can rapidly switch databases. A pragmatic perusal of animation environments shows that few animations are designed with graphics hardware that can display thousands of vectors in real time. In fact, animators are often working at a station that can handle a few vectors in real time. *twixt* allows the animators to dynamically switch the database used to draw an object. Thus the animator can have rapidly drawn frames of low complexity or slowly drawn frames of high complexity, just by replacing the surface geometry definition of an object. All parameters of an object not related to its geometry are unaltered by this replacement.

On fast graphics hardware this becomes less of a constraint. It will decrease in importance in the near future (see final section of notes).

Objects are named as described in Gomez [7, 8]. Two important points not mentioned in the paper are regular expressions and aliases. Names can contain regular expression characters like the Unix *shell* and *csh*; these characters are handled just as they would be in either of shell. The user can also define alias names, indicating that whenever *twixt* sees that name, it is to be expanded to all the objects named in the list for that alias. List elements may of course be regular expressions. Furthermore, it is not an error to include an undefined object in an alias list. *twixt* assumes that the animator will bring in that object later, when he is ready for it.

#### 4.4. Track Implementation

*twixt* implements the following tracks:

- position &  $d\vec{p}/dt$
- rotation &  $d\vec{r}/dt$
- scale &  $d\vec{s}/dt$
- attach position &  $d\vec{a}/dt$
- color &  $d\vec{c}/dt$
- shininess &  $dShininess/dt$
- transparency &  $d\vec{Transparency}/dt$
- surface geometry
- display enable flag
- attachment
- notes

##### 4.4.1. Basic geometrical transform tracks

Many of these tracks are straightforward: position, scale, color, illumination parameters. Rotation can be treated either as angles around the object's axes or as 3x3 orientation matrices. The former case is easy to implement but non-intuitive, meaning that after a few rotations, it's hard for the animator to make a direct connection between instructing the system to do a rotation and what happens on the screen. This is because the object's axes are themselves transformed, meaning that the rotation is not being applied to the original axes set but the transformed set. In the latter case, matrix interpolation was implemented using a scheme based on a question from my general examinations. This technique has been formalized as quaternion rotations [15, 16].

In addition, there are velocity tracks running alongside each primary track that has a defined derivative (*e.g.* the position track has a derivative but the display enable flag does not). The animator may address any track directly, or its derivative, or both. In the latter case, the velocity track has priority in any conflicts.

As an example, consider an animator who specifies that an object's X position is to be 0 at frame 1 and 20 at frame 24, then specifies that the X velocity is to be 10 units per second. This situation is irreconcilable. The decision to give the velocity track precedence increases the likelihood that the display function will be continuous in its derivatives.

#### 4.4.2. Hierarchy control tracks

The attach position is where a child object is attached to its parent, in terms of the parent's coordinate space. There are various ways of inserting a subtree into a tree:

##### hang

In this mode, only the offspring's position is transformed by the parent's matrix; the remainder of the matrix is calculated from the child's current display parameters. The parent's scale vector does not propagate down (see *attach* below). This mode is intended for an object that is hanging on another object, such as

a rod hanging on a pivot pin. As the pivot pin moves around, the rod must go with it, but it should pivot automatically so it remains in the same orientation.

Implementation is not difficult. To construct the offspring's matrix, first transform it's final position by its parent's matrix and place the result in the bottom row of the matrix. The upper left 3x3 is calculated as usual, with no reference to the parent matrix.

#### attach

This mode was defined by *scn\_asmblr*: parent scale values do not propagate down. It's useful for attaching light sources to other objects, since in the OSU paradigm the scale value of a light source determines its range.

#### couple

This is a conventional tree builder, where all elements of the parent's matrix propagate to the offspring nodes. This method allows a limited *squash-and-stretch* capability.

In actual implementation, *twixt* constructs matrices in a bottom to top fashion. In order to build a frame, each object's matrix must be constructed. To do this, *twixt* goes through its list of objects (which corresponds to visiting each leaf in the forest) and finds which of them



has their *newmatrix* flag set, indicating that some display parameter has changed, necessitating recalculation of the matrix. It then travels recursively up the hierarchy tree until it reaches the root of that object subtree, at which point it unwinds, constructing each object's matrix on the way down *iff* that *newmatrix* flag is set and concatenating as appropriate. No matrix is ever computed twice; the *newmatrix* flag is unset to keep that from happening. Thus each node in the tree may be visited more than once, but it won't cause extraneous matrix arithmetic.

There are two ways of removing a subtree from a tree:

*detach*

Detaches a subtree. The child object (and its children) will no longer be controlled by the parent.

*letgo*

Detaches a subtree, but maintains the current transformation as a pretransformation for future animation. This is used for objects which are related to another object for part of the animation, then detached to continue own their own way.

An example would be a hand throwing a ball. Initially, the ball would be attached to the hand during the windup. When the ball is released, it is "let-go," so from that point on in time, the hand

will have no control over the ball. However, the point at which the ball was let go determines its freeflight, so the transformation at that instant must contribute to the animation following that instant.

The attachment track controls the characteristics of the hierarchy construction. The attach position track is simply a vector showing the attach position. An attachment event simply contains a flag word showing what kind of attach (or detach) is to be performed at what time. If the event is one of the attaches (as opposed to one of the detaches) it also contains a pointer to the new parent.

#### 4.4.3. Surface geometry track

This track controls the surface geometry of an object. Object shapes are interpolated (with flexibilities previously described) between some number of defined geometries. Thus, a *blended* object has no shape it can call it's own; it is defined only when the animation is running.

The animator can freeze playback, or play back a single frame, in order to take a look at the current surface geometry. Again, to save memory, the event value field becomes a pointer to another structure that actually defines the characteristics of the actual geometry and contains the data itself.

#### 4.4.4. Notes track

Note events are just that — notes. Animators usually write down all kinds of information on their exposure sheets. Note events are the animator's notes to themselves. When the animation gets to the frame a note event belongs to, the note is printed (the animator can set a flag to enable or disable note printing).

#### 4.5. Track Manipulation

Geometric transformations can be applied to track segments just as they are to objects. Tracks can be scaled, translated, or rotated. These operations are different from changing the frame numbers in events; the former change the values in the events, the latter change the times at which the events occur. Thus the former change the control values themselves; the latter change the timing of the animation.

These track-wise operations are implemented in a simple matter: the animator specifies the track segment by frame numbers, the operation, and the operand. Rotation must be performed on vector or orientation matrix tracks; it does not make sense otherwise.

A track segment copier is provided. This together with the track transformer give the animator *instancing* capability for a track. Just as geometric primitives can be defined and transformed to build more complex objects, tracks can be defined and transformed to eliminate some of

the drudge work of animation.

As an example, consider a ball bouncing along a mirror. First the animator animates one bounce of the ball. Then he copies it two or three times, each one shifted by the appropriate time (perhaps two seconds) and the appropriate dislocation. This is the original ball animation. Then the animator makes a second instance of the ball, copies the first one's position track to the second, and multiplies the second ball's Y position track by  $-1$ . This is the reflection's animation, and the animator is done. Figure 1 shows value *vs.* time plots for this animation.

For a slightly more complex example, suppose the animation was four balls and their reflections bouncing away at right angles from a central

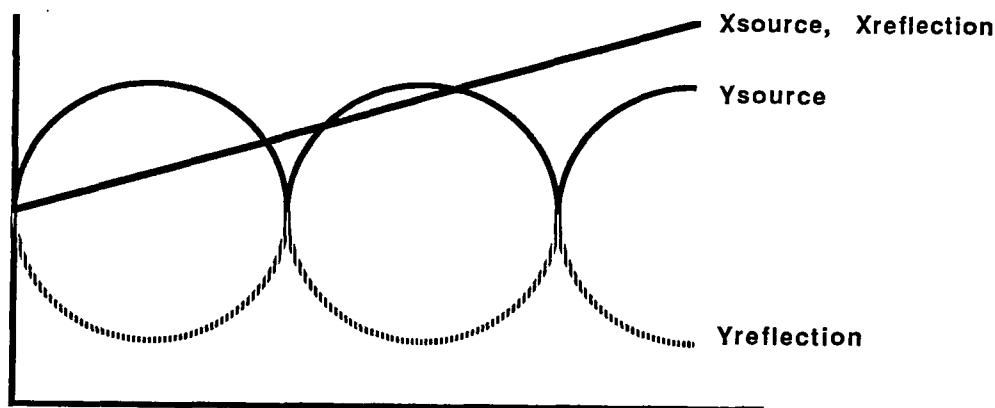



Figure 1.  
Plots of ball and reflection positions  
(Z is not important for this example)

point. As before, the animator would animation one ball and its reflection (actually, since this is already done, it's only necessary to read it in from the system). Then this duet would be copied and the copy rotated 90 degrees about the central point. This copy-rotate action is performed twice more, for a total of four balls and their reflections bouncing.

#### 4.6. Record Structures

Following are record structures showing how various entities are implemented. The '' character indicates a pointer.

##### 4.6.1. Events


| Event structure |                                                                                             |
|-----------------|---------------------------------------------------------------------------------------------|
| EventTypes      | type                                                                                        |
| ...             | value                                                                                       |
| Natural         | frame                                                                                       |
| Natural         | easeIn                                                                                      |
| Natural         | easeOut                                                                                     |
| Twerper         |  twerper |
| Logical         | dF                                                                                          |

Figure 2.

#### Event record structure

The value field has no type, because it will depend on what the event is being used for. If this structure were being implemented in PASCAL, the event type field would serve as the CASE selector for a variant record.

Whether or not the event is a velocity event can be built into the event type field or separated into its own field as is shown here. The form shown here has some runtime advantages, *e.g.* if some piece of code needs to do something to a color event, whether it's a value or a velocity value, it can work similar to this:

```
if (event.type is Color)
    Doit()
```

instead of like this:

```
if ((event.type is Color) or (event.type is ColorVelocity))
    Doit()
```

Technically, an event structure would be able to handle any kind of display parameter the user desired. Unfortunately, most compilers will simply allocate enough space for the worst case. In the case of a surface geometry definition, it would require a lot of memory. In a global context, most of the memory used would be wasted, since most events are much shorter than surface geometry definitions. Therefore it makes sense to use pointers for events that could take up a lot of memory.

#### 4.6.2. Tracks

| Track structure |                  |
|-----------------|------------------|
| Event           | → events         |
| Twerper         | → globalTwerper  |
| Event           | → derivatives    |
| Twerper         | → globalDTwerper |

Figure 3.

#### Track record structure

The event pointers are *head* pointers, *i.e.* they point to the heads of their respective lists. If a global splining function pointer is non-NULL, then the indicated function should always be used for interpolating that track; otherwise use the patched method as described previously.

An alternative form would be to have a logical flag indicating whether or not to use the global splining function. It's a matter of taste; either way should generate the same number of instructions if the NULL pointer is zero, as it is in C.

#### 4.6.3. Twerpers

| Twerper structure |        |
|-------------------|--------|
| String            | → name |
| ...               | code   |

Figure 4.

#### Interpolating function record structure

The name is used for display purposes, i.e. for telling the user what the name of the function is. It will point to something like "cubic B-spline" or "combination move," *etc.* The other field points to the code implementing that function. It will return whatever's appropriate, typically a floating point blending factor.

#### 4.6.4. Comments

The structures shown here are not the actual declarations used in the program, although they do indicate the information content required. Other fields may be useful for practical purposes. Forward and backward pointers are a help, as doubly linked list traversal is fast. Additional pointers to reduce cross list traversal or avoid indirected lookup also save time. Theoretically, they're not necessary, but faster is better.

Obviously there's more to writing an animation system than what's discussed here. These concepts, however, form the basis around which *twixt* is written. There is a lot more that could be described, but that would be outside the scope of these particular notes. Some additional references along these lines are my dissertation [9] and the user manual [8].



## 5. Epilogue

An extension to the idea of modifying tracks is to transform them with modifying functions, *i.e.* to filter the display function through time. This would be one way of providing character. After designing a walk cycle, the animator would apply a modifier to provide a particular kind of walk, *e.g.* a limp. There are analogies between this and the NYIT motion postprocessors and Perlin's pixel stream editor [12].

Current developments in fast 3-D raster display systems will not have as much of an impact as advanced user capabilities, because fast hardware is not the hard problem in computer animation. Animators generally desire to see frames of high complexity in full color with advanced surface modeling techniques (note that this is different from actual contemporary situations); advanced 3-D systems generally work only with polygons and simple illumination calculations. The bandwidth required for complex 3-D imagery far exceeds the capability of any current or planned hardware system. Thus the major advances in computer animation will come not from better display units, but from more advanced capabilities available to the animator.

Building an animation system is a nontrivial task. Doing it requires implementations of techniques from all aspects of computer science. It's better to view an animation system as a tool, since its function is to be used, rather than to be an end in itself. Much of the system's success will come from its users' imagination. But it has to provide them with the appropriate levels of

abstraction and the appropriate hierarchy of complexities, where "appropriate" is the nebulous quantity indicating it's not overbearing in normal use but smart enough to help get the job done.

Developers and animators must remain in constant contact over the lifetime of an animation system; otherwise the it will end up being skewed towards the group that built it. The design and development of an animation system should be seen as a symbiotic task between the "technical" types and the "artist" types.

## 6. References

1. Baecker, Ronald M, "Picture-driven animation," in *Interactive Computer Graphics*, ed. Herbert Freeman, IEEE Computer Society(1980). Originally published in *Conference Proceedings, Spring Joint Computer Conference*, AFIPS, 1969
2. Bergeron, Daniel and et, al, *Tony di Peltrie, ie !" Animation."* Animation. 1985.
3. Catmull, Edwin, "The Problems of Computer-Assisted Animation," *Computer Graphics* 12(3)(August 1978).
4. Crow, Franklin C, "A More Flexible Image Generation Environment," *Computer Graphics* 16(3) pp. 9-18 SIGGRAPH-ACM, (July 1982).
5. DeFanti, Thomas A, "The Graphics Symbiosis System -- An Interactive Minicomputer Graphics Language Designed for Habitability and Extensibility," Ph. D. Dissertation, The Ohio State University (March 1973).
6. Donkin, John, *Trash*, Ohio State University CGRG (1984). Animation.
7. Gomez, Julian E, "Twixt: A 3D Animation System," *Computers and Graphics* 9(9) pp. 291-298 Pergamon Press Ltd., (1985). Reprinted from *Proceedings of Eurographics '84*.
8. Gomez, Julian E, *twixt user manual*, Computer Graphics Research Group, The Ohio State University (1985).
9. Gomez, Julian E, *Computer Display of Time Variant Functions*, The Ohio State University (1985). Ph.D. dissertation
10. Knuth, Donald E., *The Art of Computer Programming Volume 2: Seminumerical Algorithms*, Addison-Wesley Publishing Co., Reading, Mass. (1969).
11. Lasseter, John, *Luxo, Jr.*, Pixar, Inc. (1986). Animation.
12. Perlin, Ken, "An Image Synthesizer," *Computer Graphics* 19(3)(July 1985).
13. Reynolds, Craig W, "Computer Animation with Scripts and Actors," *Computer Graphics* 16(3) pp. 289-296 SIGGRAPH-ACM, (July 1982).
14. Schlag, John F., "Eliminating the Dichotomy Between Scripting and Interaction," in *Proc. Graphics Interface '86*, (May 1986).
15. Shoemake, Ken, "Animating Rotation with Quaternion Curves," *Computer Graphics* 19(3)(July 1985).
16. Shoemake, Ken, "Quaternion Calculus and Fast Animation," in *Tutorial Notes: Computer Animation: 3-D Motion Specification and Control*, ACM SIGGRAPH(July 1987).

17. Smith, Alvy Ray, "Spline Tutorial Notes," in *Tutorial Notes: Computer Animation*, SIGGRAPH(1984).
18. Teitelman, Warren, "The Cedar Programming Environment: A Midterm Report and Examination," CSL-83-11, Xerox PARC (June 1984).
19. Teitelman, Warren, "A Tour Through Cedar," *IEEE Software* 1(2) pp. 44-73 (April 1984).
20. Thomas, Frank and Johnston, Ollie, *Disney Animation: The Illusion of Life*, Abbeville Press, New York (1981).
21. VanBaerle, Susan and Kingsbury, Doug, *Snoot and Muttly*, Ohio State University CGRG (1984). Animation.
22. Wedge, Chris, *Tuber's Two Step*, Ohio State University CGRG (1985). Animation.
23. Wilhelms, Jane, "Towards Automatic Motion Control," in *Tutorial Notes: Computer Animation: 3-D Motion Specification and Control*, 1986
24. Zeltzer, David, Gomez, Julian, and MacDougal, Paul, "A Tool Set for 3-D Computer Animation," in *Tutorial Notes: Introduction to Computer Animation*, SIGGRAPH(1984).
25. Zeltzer, David, "Toward An Integrated View of 3-D Computer Animation," in *Tutorial Notes: Introduction to Computer Animation*, SIGGRAPH(1984).